Table of Contents

[Introduction 3](#_Toc467958046)

[Design and Analysis of the algorithm 3](#_Toc467958047)

[Implementation and testing of the algorithm 8](#_Toc467958048)

[Validation of pre-implementation analysis of program 10](#_Toc467958049)

[Epilogue 11](#_Toc467958050)

[Appendix A: Program Listing 12](#_Toc467958051)

[Appendix B: Output 12](#_Toc467958052)

[Code Output: 12](#_Toc467958053)

[Screenshots: 17](#_Toc467958054)

# Introduction

Thanks Giving is well known for family gatherings and everyone tries to plan a trip to meet their families. During this week, almost every individual plan for a car trip and as a traveler we always wanted to choose the shortest path to reach our destination. As we start moving on this shortest path we find that we are not the only one on this route. Even many individuals consider this route as the shortest path to reach their destination and eventually the path is congested. Now the shortest path became a congested path and because all cars are using the shortest path and the shortest path for a congested road is different from only one individual on this shortest path. This the main problem statement for our Project. To solve this problem, we need to design an algorithm to find the shortest path even when the path is congested.

# Design and Analysis of the algorithm

For this problem stated above we can estimate or calculate the load of that particular start and end point of a route based on the flow of that particular connected graph/roads. This flow is based on average number of cars at that particular road for start and end points. But this approach is very poor and congestion happens all the time. For Example, this thanksgiving holiday 2016, Los Angeles roads are completely flooded with cars where we can see only white line and red line if we see from overhead chopper, where there are headlights and tail lights of cars.

To reduce this congestion, we have considered 3 approaches differently and had group discussion on each approach and finalized a new approach by combining two of those proposed approaches. As mentioned in the project document provided by professor, One approach is one car at one time into system and iterate until all cars are accommodated. This approach is resulting in iterating many number of times until all cars are accommodated. This will be a lot of time consuming and cars can use another approach to reach destination. Second approach is to accept and allocate one car at one time for each s-t pair and iterate all cars are accommodated. This approach will reduce time when compared to first approach but checking for each s-t pair will create another problem with priority of which s-t pair should we start with. Another approach is to accept and allocate all s-t traffic such that first s-t which are one hop, next two hops and so on. With this approach if s-t pair with one hop are done first and when comes to s-t pairs with two hops, then already first 2 nodes in two hop s-t pair are traversed in first iteration, this iteration is not needed for that s-t pair.

So we decided a new approach, where first we will accept and allocate one car at one time for each s-t pair of one hop, followed by one car at one time by all s-t pairs which are two hops away and so on. This is reducing the time and when compared the actual edge delay this is somewhere within 10% of predicted edge delay.

Following is the algorithm designed for the approach we have chosen:

**Algorithm 1** Calculate Shortest Path with Congestion

**procedure** ShortestPathAlgo()

**try do**

*filename*=*read from command line;*

*f = open(filename, 'read-only mode');*

**except** IOError **do**

*print* (‘Incorrect Filename’);

**end procedure/exit from program**

*f1 = f.read().splitlines();*

*f2 = [i.split(',') for i in f1];*

*INF=*999999999

*start*=f2[0][1]

*dest*=f2[0][2]

*n=*f2[0][0]

*Edge*=[*n*][*n*]=INF

*ShortEdge*=[*n*][*n*]

*AllPairPath*=[*n*][*n*]

*HopCount*=[*n*][*n*] = 0

*Capacity*=[*n*][*n*]

*Load*=[*n*][*n*]

*actualEdgeDelay*=[*n*][*n*]

*actualPathDelay*=[*n*][*n*]

**for** each line *i* in *f2* **do**

*code*=*f2*[*i*][0]

*from*=*f2*[*i*][1]

*to*=*f2*[*i*][2]

*value*=*f2*[*i*][3]

**if** code=='E' **then**

*Edge*[*from*][*to*] = *value*

**else if** code=='C' **then**

*Capacity*[*from*][*to*] = *value*

**end if**

**end for**

//Floyd Warshall Algorithm for all pairs shortest path

**for** k in range(1,n+1) **do**

**for** i in range(1,n+1) **do**

**for** j in range(1,n+1) **do**

**if** *ShortEdge*[*i*][*k*] + *ShortEdge* [*k*][*j*]< *ShortEdge* [*i*][*j*] **then**

*ShortEdge* [*i*][*j*] = *ShortEdge* [*i*][*k*] + *ShortEdge* [*k*][*j*]

*AllPairPath* [*i*][*j*] = *AllPairPath* [*i*][*k*][*-1*] + *AllPairPath* [*k*][*j*]

**end if**

**end for**

**end for**

**end for**

*print*(‘Predicted Path Length after performing all-pairs-shortest-paths for a single car is :’, *ShortEdge*)

*print*(‘Actual all-pairs-shortest-paths for a single car is:’)

**for** *elem* in *AllPairPath* **do**

**for** *y* in *AllPairPath* [*elem*] **do**

*count*=*len*(*AllPairPath* [*elem*][*y*])-1

*HopCount*[*elem*][*y*]=*count*

*print*(*AllPairPath* [*elem*][*y*], "*\t*")

**end for**

**end for**

*maxhop*=0

*print*(‘Hop Count for :’)

**for** *elem* in *HopCount* **do**

**for** *y* in *HopCount*[*elem*] **do**

**if** *maxhop* < *HopCount*[*elem*][*y*] **then**

*maxhop*=*HopCount*[*elem*][*y*]

**end if**

**if** *HopCount*[*elem*][*y*]==1 **then**

*ShortEdge* [*elem*][*y*]=1

**end if**

*print*(*HopCount*[*elem*][*y*],'*\t*')

**end for**

**end for**

**for** *i* in *range*(0,n) **do**

**for** j in range(0,n) **do**

**if** *Capacity*[*i*][*j*]!=INF:

**if** *Capacity*[*i*][*j*]>=*Load*[*i*][*j*]:

*Cap*=*Capacity*[*i*][*j*]+1

*actualEdgeDelay* [*i*][*j*]= *Cap*\**Edge*[*i*][*j*] / (*Cap*-*Load*[*i*][*j*])

**else**

*actualEdgeDelay* [*i*][*j*]=*INF*

**end if**

**else**

*actualEdgeDelay* [*i*][*j*]=*INF*

**end if**

**end for**

**end for**

**for** *o* in *range*(1,*maxhop*) **do**

**for** *x* in *range*(0,*n*) **do**

**for** *y* in *range*(0,*n*) **do**

*Load*[*x*][*y*]=0

**for** *i* in *range*(0,*n*) **do**

**for** *j* in *range*(0,*n*) **do**

**if** *o*+1 < = *HopCount*[i][j] **then**

*u*= *AllPairPath* [*i*][*j*][*o*]

*v*= *AllPairPath* [*i*][*j*][*o*+1]

**if** *len*(*AllPairPath* [u][v])==2 **then**

*Load*[u][v] = 1

**for** *i* in *range*(0,*n*) **do**

**for** *j* in *range*(0,*n*) **do**

*Cap* **=** *Capacity*[i][j]+1

*actualEdgeDelay* [i][j]= (*Cap* \* *actualEdgeDelay* [i][j])/(*Cap*-*Load*[i][j])

**end for**

**end for**

*print*(‘Actual Edge Delay G Matrix for our designed approach:’, *actualEdgeDelay*)

**for** *i* in *range*(0,*n*) **do**

**for** *j* in *range*(0,*n*) **do**

**if** *HopCount*[*i*][*j*] = =1 **then**

*actualPathDelay*[i][j]=*actualEdgeDelay*[i][j]

**else if** *HopCount*[*i*][*j*] > 1 **then**

**for** *k* in *range*(0,*HopCount*[*i*][*j*]) **do**

*ab*=*AllPairPath* [i][j][k]

*cd*= *AllPairPath* [i][j][k+1]

*actualPathDelay*[i][j]= *actualPathDelay*[i][j]+*actualEdgeDelay*[ab][cd]

**end for**

**end if**

**end for**

**end for**

*print*(‘Actual Path Delay G Matrix for our designed approach:’, *actualPathDelay*)

*ShortestPredEdgeLength*=*Edge*[*start*-1][*dest*-1]

*DesShortestPredEdgeLength*=*actualEdgeDelay*[*start*-1][*dest*-1]

**if** *ShortestPredEdgeLength* = = *INF* **do**

*ShortestPredEdgeLength*="\'NA\'"

**if** *DesShortestPredEdgeLength* = = *INF* **do**

*DesShortestPredEdgeLength*="\'NA\'"

*print*(‘Shortest Predicted Path Length is:’,*ShortEdge*[*start*][*dest*], ‘and Actual Path length is:’,*actualPathDelay*[*start*-1][*dest*-1],’from’,*start*,’to’,*dest*)

*print*(‘Shortest Predicted Edge Length:",*ShortestPredEdgeLength*,"and Actual Edge length:",*DesShortestPredEdgeLength*,"from",*start*,"to",*dest*)

*print*(‘The hop-count of the path between",*start*,"and",*dest*,"is:",*HopCount*[*start*][*dest*])

**end procedure**

In the above algorithm we have used Floyd Warshall algorithm for calculating all-pairs-shortest paths with little bit of changes by calculating the path for that particular s-t pair at the same time. Also we have used dictionaries while coding in python version 3.5.2 to reduce the traversal time and complexity while printing the matrices/dictionaries/arrays.

After calculating shortest paths, we calculate load based on our approach where one car at one time for each s-t pair of one hop, followed by two hops and so on. For this first we calculated load for all s-t pairs of one hop. Later iterated for 2 hops s-t pairs where we check the path existence and change the load for every iteration and so on. Finally, after completing all hops for s-t pairs we will have the matrix with actual edge delays. We calculated actual path delays based on actual edge delay values and can get values for any given source and destination node.

# Implementation and testing of the algorithm

For this problem we decided to use Python version 3.5.2 as our programming language, because it is an interpreted language unlike C++ which uses compiler. As we already know python is an indentation based programming language where traversing through lines and files will make easy. Also we can translate our algorithm to program very easily when compared to huge programs like Java. Also, in this problem since we are using s-t pairs approach for graphs, python will represent them in the form of lists and dictionaries which will ease the programmer’s effort while performing multiple traversals and calculations. Also we can traverse to single individual element of a list/dictionary, which is necessary to our approach, where we have to compare and check each path node of an s-t pair with more than two hops away. In python we can write the whole code in single line with indentation which will make the code more readable and legible.

For testing the algorithm and program we have used the following test cases:

|  |  |  |
| --- | --- | --- |
| Test Case | Expected Result | Actual Result |
| Verify whether the input file is read and n, start and destination values are obtained | Check file read without error and n, start and destination variables should have values from files. | Program successfully reads the input values from the file. n, start and destination are obtained. |
| Verify whether the Edge, Capacity and Flow are properly initialized | Check whether Edge, Capacity and Flow are properly initialized able printed | Program successfully initializes Edge , Capacity and Flow matrix and prints them in the command prompt. |
| Verify whether the hop count matrix is correctly updated. | Check that hop counts are calculated and able to print | Observed that the program calculates and prints the exact hop count in the command prompt. |
| Verify that the Floyd Warshall algorithm gives the appropriate shortest path between the nodes | Check that the shortest paths for small test input were calculated correctly. | Program successfully gives output for the calculation of shortest paths using the Floyd Warshall algorithm. |
| Verify that the Floyd Warshall algorithm gives the exact path for the determined shortest path | Check that paths are updated exactly for taken small test input. | Program Successfully prints the appropriate shortest path of the nodes in the algorithm. |
| Verify the calculation of the Actual edge delay using the implemented algorithm. | Verified that the implemented algorithm gives actual edge delay values correctly. | Successful output is printed in the Actual edge delay calculations. |
| Verify the calculation of the Actual path delay in the implemented algorithm. | Verified that the implemented algorithm calculates the Actual path delay appropriately. | Observed that the program calculates the actual path delay values correctly. |
| Verify the percentage calculation of the Predicted path delay and Actual path delay | Verified that the Percentage calculations are appropriate and computed correctly. | Program successfully computes the percentage calculations and are printed in the command prompt. |
| Verify that the Predicted edge length and Actual edge length are calculated and printed correctly for given source and destination | Verified that the Predicted edge length and Actual edge length are printed correctly for given source and destination | Program successfully calculates and prints the Predicted edge length and Actual edge length for given source and destination |
| Verify that the Predicted path length and Actual path length are calculated and printed correctly for given source and destination | Verified that the Predicted path length and Actual path length are printed correctly for given source and destination | Program successfully calculates and prints the Predicted path length and Actual path length for given source and destination |
| Verify that the hop count is printed for given source and destination | Verified that the hop count is printed for given source and destination | Program successfully verifies the hop count and is printed for given source and destination. |

While programming in python, we have used dictionaries while calculating shortest edge paths, shortest paths route and hop count so that we can traverse through each element whenever they are needed without using two for loops(i,j) for traversal. Also, for every 2-dimensional matrix we used list representation so that it will be easy while appending string or path values to current values in the list.

# Validation of pre-implementation analysis of program

Here we have used time library to calculate the time taken to execute the program.

**Line 1:** import time

**Line 5:** startTime = time.time()

##Program Block Code

**Line 400:** endTime = time.time()

**Line 401:** print ("\n",round(endTime - startTime,2)," seconds -> Time taken to execute the file")

The key-and-basic operation of this algorithm is to calculate the actual path delay (G) for given n. G is calculated using this snippet of algorithm:

**if** maxhop>1:

**for** k in range(0,maxhop-1):

**for** i in range(0,n):

**for** j in range(0,n):

**if** C[i][j]!=INF:

**if** C[i][j]>=newL[i][j]:

ActualG[i][j]=round(((C[i][j]+1)\*ActualG[i][j])/((C[i][j]+1)-newL[i][j]),2)

**else**:

ActualG[i][j]=INF

**end if**

**else**:

ActualG[i][j]=INF

**end if**

**end for**

**end for**

**end for**

**end if**

Actual Edge Delay is the final actual path delay due to congested path in this algorithm. And this value is calculated for each i and each j, which are the number of rows and columns of the edge delay matrix, multiplied by the maximum number of hop counts. Let the maximum hop counts be ‘m’. Hence the time complexity becomes O(m.n2).

The best case happens for higher n values happens when max hop count for the whole system is 1. The best case time complexity would be n2. The time complexity increases as maximum hop count increases as there will be more load on

Below is a table representing this predicted time and the actual time taken by the code to implement for this part of the algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| n | m | Predicted | Actual |
| 6 | **3** | 108 | 1.029 msec |
| 15 | **5** | 1125 | 3.036 msec |
| 10 | **4** | 400 | 1.041 msec |

Here we observe that as n increases the ratio in predicted and actual times are consistent with each other. Though for smaller test cases this pattern is not followed. If we double the input the predicted is increased 4 times whereas the actual time has very less difference. But when increased around 3 times, the actual time significantly increased thrice. This can be because of the environment of the machine working and the availability of memory space to perform basic operations.

# Epilogue

In the given problem, we know edge, flow and capacity matrices are taken from file and calculating edge and path delays based on a formula. But we haven’t consider the time variable since the capacity is said based on time, for example capacity can be 10 cars per hour for that s-t edge. If we consider that probability, then after allowing maximum capacity cars for that particular s-t pair then we should stop the upcoming cars for that s-t pair, and we should calculate the next best route for the destination since the s-t pair intended to go is closed because of reaching its maximum capacity.

This project helped a lot in thinking in all possibilities for a particular problem, and to think of unthinkable situations while analyzing the problem.

# Appendix A: Program Listing

 => PredictingShortestPathWithCongestion.py

The file attached above is the python file (\*.py) for this project.

As we already mentioned in implementation of algorithm part, we are using python of version 3.5.2. Here it runs Python compiler automatically, but we can’t notice like in C++ program compiling. But \*.pyc and \*.pyo files are generated for the modules we import. We used Windows 10 operating system, with intel processor with basic python ide 3.5.2 version.

We can execute the file by keeping the input text file in the same path of \*.py file. Then using command line “python <filename.py>” or we can use any python IDE like pyCharm from Jetbrains Idea to execute the code but the input text file should be in the same folder of program \*.py.

# Appendix B: Output

After executing the code in python the following are the outputs for the test file with 6 nodes.

## Code Output:

Enter the file Name:input.txt

Given Edge Matrix in file:

[0, 7, 999999999, 7, 999999999, 9]

[999999999, 0, 5, 999999999, 10, 3]

[9, 10, 0, 8, 4, 6]

[9, 4, 2, 0, 999999999, 999999999]

[3, 5, 10, 10, 0, 999999999]

[999999999, 5, 8, 10, 999999999, 0]

Given Flow Matrix in file:

[0, 9, 11, 12, 8, 12]

[18, 0, 15, 10, 17, 18]

[17, 18, 0, 14, 10, 10]

[17, 8, 10, 0, 17, 18]

[15, 9, 12, 14, 0, 16]

[18, 16, 15, 8, 9, 0]

Given Capacity Matrix in file:

[0, 13, 999999999, 33, 999999999, 20]

[999999999, 0, 67, 999999999, 5, 55]

[5, 5, 0, 32, 134, 17]

[23, 34, 55, 0, 999999999, 999999999]

[68, 47, 20, 14, 0, 999999999]

[999999999, 16, 44, 16, 999999999, 0]

Predicted Path Length after performing all-pairs-shortest-paths for a single car is :

0 7 9 7 13 9

12 0 5 13 9 3

7 9 0 8 4 6

9 4 2 0 6 7

3 5 10 10 0 8

15 5 8 10 12 0

Actual all-pairs-shortest-paths for a single car is:

[1] [1, 2] [1, 4, 3] [1, 4] [1, 4, 3, 5] [1, 6]

[2, 3, 5, 1] [2] [2, 3] [2, 3, 4] [2, 3, 5] [2, 6]

[3, 5, 1] [3, 5, 2] [3] [3, 4] [3, 5] [3, 6]

[4, 1] [4, 2] [4, 3] [4] [4, 3, 5] [4, 2, 6]

[5, 1] [5, 2] [5, 3] [5, 4] [5] [5, 2, 6]

[6, 3, 5, 1] [6, 2] [6, 3] [6, 4] [6, 3, 5] [6]

Hop Count for :

0 1 2 1 3 1

3 0 1 2 2 1

2 2 0 1 1 1

1 1 1 0 2 2

1 1 1 1 0 2

3 1 1 1 2 0

Actual Edge Delay G Matrix for approach provided:

[0.0, 19.6, 999999999, 79.33, 999999999, 21.0]

[999999999, 0.0, 42.5, 999999999, 10.0, 42.0]

[9.0, 10.0, 0.0, 29.33, 180.0, 13.5]

[30.86, 15.56, 11.2, 0.0, 999999999, 999999999]

[207.0, 48.0, 23.33, 150.0, 0.0, 999999999]

[999999999, 85.0, 120.0, 18.89, 999999999, 0.0]

Actual Path Delay G Matrix for approach provided:

[0, 19.6, 90.53, 79.33, 270.53, 21.0]

[429.5, 0, 42.5, 71.83, 222.5, 42.0]

[387.0, 228.0, 0, 29.33, 180.0, 13.5]

[30.86, 15.56, 11.2, 0, 191.2, 57.56]

[207.0, 48.0, 23.33, 150.0, 0, 90.0]

[507.0, 85.0, 120.0, 18.89, 300.0, 0]

Actual Edge Delay G Matrix for our designed approach:

[0.0, 7.54, 999999999.0, 7.21, 999999999.0, 9.45]

[999999999.0, 0.0, 5.07, 999999999.0, 10.0, 3.11]

[9.0, 10.0, 0.0, 8.51, 4.09, 6.35]

[9.39, 4.12, 2.08, 0.0, 999999999.0, 999999999.0]

[3.13, 5.22, 10.5, 10.71, 0.0, 999999999.0]

[999999999.0, 5.31, 8.18, 10.62, 999999999.0, 0.0]

Actual Path Delay G Matrix for our designed approach:

[0, 7.54, 9.29, 7.21, 13.38, 9.45]

[12.29, 0, 5.07, 13.58, 9.16, 3.11]

[7.22, 9.31, 0, 8.51, 4.09, 6.35]

[9.39, 4.12, 2.08, 0, 6.17, 7.23]

[3.13, 5.22, 10.5, 10.71, 0, 8.33]

[15.4, 5.31, 8.18, 10.62, 12.27, 0]

Actual and Predicted path Difference (in percentage):

7.71 3.22 3.0 2.92 5.0

2.42 1.4 4.46 1.78 3.67

3.14 3.44 6.38 2.25 5.83

4.33 3.0 4.0 2.83 3.29

4.33 4.4 5.0 7.1 4.12

2.67 6.2 2.25 6.2 2.25

Shortest Predicted Path Length is: 8 and Actual Path length is: 8.18 from 6 to 3

Shortest Predicted Edge Length: 8 and Actual Edge length: 8.18 from 6 to 3

The hop-count of the path between 6 and 3 is: 1

2.57 seconds -> Time taken to execute the file

## Screenshots:

